## Java中日期转换

### 1.util.Date-->long

 Date date = new Date();

 long time = date.getTime();

 System.out.println(time);*//1652058017799*

### 2.long-->util.Date

long currentTimeMillis = System.currentTimeMillis();

Date date = new Date(currentTimeMillis);

System.out.println(date);*//Mon May 09 09:07:02 CST 2022*

### 3.util.Date-->String Date

Date  d = new Date();

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String format = sdf.format(d);

System.out.println(format);*//2022-05-09 09:19:45*

### 4.String Date-->util.Date

String date = "2022-05-09 09:19:45";

 SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

 Date parse = sdf.parse(date);

 System.out.println(parse);*//Mon May 09 09:19:45 CST 2022*

### 5. 对日期进行封装

public class DateUtil {

public static final String FMT\_DATE ="yyyy-MM-dd";

public static final String FMT\_TIME ="HH:mm:ss";

public static final String FMT\_DATETIME ="yyyy-MM-dd HH:mm:ss";

public static final String FMT\_DATETIME\_SHORT ="yyyy-MM-dd HH:mm";

public static final String FMT\_YEAR ="yyyy";

public static final String FMT\_MONTH ="MM";

public static final String FMT\_DAY ="dd";

public static final String FMT\_MONTH\_DAY ="MM-dd";

*/\*\**

 \* util.Date转换为对应的long类型日期

 \* @param date util.Date

 \* @return  long类型日期

 \*/

public static long getLongDate(Date date) {

 return date.getTime();

}

*/\*\**

 \* long类型日期转为util.Date

 \* @param date long类型日期

 \* @return 返回util.Date

 \*/

public static Date getUtilDate(long date) {

 return new Date(date);

}

*/\*\**

 \* util.Date转换为指定格式的String Date

 \* @param date util.Date

 \* @param pattern  指定时间格式

 \* @return 返回String类型的日期

 \*/

public static String getStrDate(Date date,String pattern) {

 SimpleDateFormat sdf = new SimpleDateFormat(pattern);

 String format = sdf.format(date);

 return format;

}

*/\*\**

 \* String Date转换为 util.Date

 \* @param date String格式时间

 \* @param pattern 时间格式

 \* @return  返回util.Date日期

 \*/

public static Date getStrToDate(String date,String pattern) {

 SimpleDateFormat sdf = new SimpleDateFormat(pattern);

 Date parse = null;

 try {

  parse = sdf.parse(date);

 } catch (ParseException e) {

*// TODO Auto-generated catch block*

  e.printStackTrace();

 }

 return parse;

}

}

## Java中日期计算

在Java中需要做时间的计算的时候可以考虑Calendar；

### Calendar对象

*//1.得到对象*

Calendar instance = Calendar.getInstance();

*//2.instance.get(常量) 以整数的方式得到当前日期中的某一部分*

int year = instance.get(Calendar.YEAR);

System.out.println(year);*//获取年份*

int month = instance.get(Calendar.MONTH)+1;*//获取月份，范围0-11*

System.out.println(month);*//5*

int day = instance.get(Calendar.DAY\_OF\_MONTH);*//获取日期部分*

System.out.println(day);*//9*

int hour = instance.get(Calendar.HOUR);*//获取小时*

System.out.println(hour);*//10*

int minute = instance.get(Calendar.MINUTE);*//获取分*

System.out.println(minute);*//21*

int second = instance.get(Calendar.SECOND);*//获取秒*

System.out.println(second);*//20*

*//获取当前日期这一天在当前年的第多少天;*

int h\_day = instance.get(Calendar.DAY\_OF\_YEAR);

System.out.println(h\_day);

*//获取当前日期是当前月的第几天*

int h\_month = instance.get(Calendar.DAY\_OF\_MONTH);

System.out.println(h\_month);

*//获取当前日期是当前周的第几天(默认周日为第一天)*

int h\_week = instance.get(Calendar.DAY\_OF\_WEEK);

System.out.println(h\_week);

*//返回long类型的以毫秒为单位的*Calendar时间

int timeInMillis= instance.getTimeInMillis();

System.out.println(timeInMillis);

### 1.计算两个日期之间相差多少秒

*/\*\**

 \* 计算两个日期之间相差多少秒

 \* @param sDate 较小日期

 \* @param eDate 较大日期

 \* @return

 \*/

public static long getDifferOfSecond(Date sDate,Date eDate){

 Calendar sCalendar = Calendar.getInstance();

 Calendar eCalendar = Calendar.getInstance();

 eCalendar.setTime(eDate);

 sCalendar.setTime(sDate);

 return ((eCalendar.getTimeInMillis()-sCalendar.getTimeInMillis())/1000);

}

### 2.计算两个日期之间相差多少分钟

*/\*\**

 \* 计算两个日期之间相差多少分钟，

 \* @param sDate 较小日期

 \* @param eDate 较大日期

 \* @return  分钟数，不足1分钟返回0

 \*/

public static long getDifferOfMinute(Date sDate,Date eDate){

 Calendar sCalendar = Calendar.getInstance();

 Calendar eCalendar = Calendar.getInstance();

 eCalendar.setTime(eDate);

 sCalendar.setTime(sDate);

 return ((eCalendar.getTimeInMillis()-sCalendar.getTimeInMillis())/(1000\*60));

}

### 3.计算两个日期之间相差多少小时

*/\*\**

 \* 计算两个日期之间相差多少小时

 \* @param sDate 较小日期

 \* @param eDate 较大日期

 \* @return  小时数，不足1小时返回0，

 \*/

public static long getDifferOfHour(Date sDate,Date eDate){

 Calendar sCalendar = Calendar.getInstance();

 Calendar eCalendar = Calendar.getInstance();

 eCalendar.setTime(eDate);

 sCalendar.setTime(sDate);

 return ((eCalendar.getTimeInMillis()-sCalendar.getTimeInMillis())/(1000\*60\*60));

}

### 4.计算两个日期之间相差的天数

*/\*\**

 \* 计算两个日期之间相差的天数

 \* @param sDate 较小的时间

 \* @param eDate  较大的时间

 \* @return 相差天数 不足1天返回0

 \* @throws ParseException

 \*/

public static int getDifferOfDay(Date sDate,Date eDate) throws ParseException  {

    Calendar cals = Calendar.getInstance();

    Calendar cale = Calendar.getInstance();

    cals.setTime(sDate);

    cale.setTime(eDate);

*///设置时间为0时*

    cals.set(Calendar.HOUR\_OF\_DAY,0);

    cals.set(Calendar.MINUTE, 0);

    cals.set(Calendar.SECOND, 0);

    long stime = cals.getTimeInMillis();

    cale.set(Calendar.HOUR\_OF\_DAY,0);

    cale.set(Calendar.MINUTE, 0);

    cale.set(Calendar.SECOND, 0);

    long etime = cale.getTimeInMillis();

    long between\_days=(etime-stime)/(1000\*3600\*24);

   return Integer.parseInt(String.valueOf(between\_days));

}

### 5.计算两个日期相差的月数

*/\*\**

 \* 计算两个日期相差的月数

 \* @param sDate 较小的日期

 \* @param eDate 较大的日期

 \* @return 相差月数，不足1月返回0

 \* @throws ParseException

 \*/

public static int getDifferOfMonth(Date sDate,Date eDate) throws ParseException  {

 Calendar sCalendar = Calendar.getInstance();

 Calendar eCalendar = Calendar.getInstance();

 sCalendar.setTime(sDate);

 eCalendar.setTime(eDate);

 int year =eCalendar.get(Calendar.YEAR)-sCalendar.get(Calendar.YEAR);

*//开始日期若小月结束日期*

 if(year<0){

 year=-year;

 return year\*12+sCalendar.get(Calendar.MONTH)-eCalendar.get(Calendar.MONTH);

 }

 return year\*12+eCalendar.get(Calendar.MONTH)-sCalendar.get(Calendar.MONTH);

}

### 6.计算两个日期相差的年数

*/\*\**

 \* 计算两个日期相差的年数

 \* @param sDate 较小日期

 \* @param eDate 较大日期

 \* @return 相差年数，不足1年返回0

 \* @throws ParseException

 \*/

public static int getDifferOfYear(Date sDate,Date eDate) throws ParseException  {

 Calendar cal1 = Calendar.getInstance();

 Calendar cal2 = Calendar.getInstance();

 cal2.setTime(eDate);

 cal1.setTime(sDate);

 int sYear = cal1.get(Calendar.YEAR);

 int eYear = cal2.get(Calendar.YEAR);

 int year = eYear - sYear;

 return   year;

}

### 7.获取本周的第一天

*/\*\**

     \* 获取本周的第一天（以星期日为一周的第一天）

     \* @return Date 返回本周第一天的日期

  \* @throws ParseException

     \* \*\*/

    public static Date getWeekStart() throws ParseException{

        Calendar cal=Calendar.getInstance();

        cal.add(Calendar.WEEK\_OF\_MONTH, 0);

        cal.set(Calendar.DAY\_OF\_WEEK, 1);

        Date time=cal.getTime();

        return time;

    }

### 8.获取本周的最后一天

*/\*\**

     \* 获取本周的最后一天（以星期六为一周的最后一条）

     \* @return Date

     \* @throws ParseException

     \* \*\*/

    public static Date getWeekEnd() throws ParseException{

        Calendar cal=Calendar.getInstance();

        cal.set(Calendar.DAY\_OF\_WEEK, cal.getActualMaximum(Calendar.DAY\_OF\_WEEK));

        cal.add(Calendar.DAY\_OF\_WEEK, 0);

        Date time=cal.getTime();

        return time;

    }

### 9.获取上周的星期一的时间

*/\*\**

 \* 获取上周的星期一的时间

 \* @return

 \*/

public static Date getPreviousMonday() {

    Calendar cal = Calendar.getInstance();

*// 将每周第一天设为星期一，默认是星期天*

    cal.setFirstDayOfWeek(Calendar.MONDAY);

    cal.add(Calendar.DATE, -1 \* 7);

    cal.set(Calendar.DAY\_OF\_WEEK, Calendar.MONDAY);

    cal.set(Calendar.HOUR\_OF\_DAY, 0);

    cal.set(Calendar.MINUTE, 0);

    cal.set(Calendar.SECOND, 0);

    cal.set(Calendar.MILLISECOND, 0);

    return cal.getTime();

}

### 10.判断两个日期是否为同一天

*/\*\**

 \* 判断是否为同一天

 \* @param date1

 \* @param date2

 \* @return  同一天返回true，否则返回false

 \*/

public static boolean sameDay(Date date1,Date date2) {

 Calendar cal1 = Calendar.getInstance();

 Calendar cal2 = Calendar.getInstance();

 cal1.setTime(date1);

 cal2.setTime(date2);

 boolean sameDay = cal1.get(Calendar.YEAR) == cal2.get(Calendar.YEAR) &&

     cal1.get(Calendar.DAY\_OF\_YEAR) == cal2.get(Calendar.DAY\_OF\_YEAR);

 return sameDay;

}

### 11.获取当前日期前后N天的日期

*/\*\**

 \* 获取当前日期前后N天的日期

 \* @param day 正数：表示N天后,负数表示N天前

 \* @return 返回N天前后日期

 \*/

public static Date getCurrentTimeAddDay(Integer day){

 Calendar calendar = Calendar.getInstance();

 calendar.setTime(new Date());

 calendar.add(Calendar.DATE, day);

 return calendar.getTime();

}

## Java中如何操作较大数值的计算？

### 一、实际开发中遇到的问题

操作的数值很大，首先想到的是double,如果两个数计算超过了Double的范畴需要怎么处理？

### 二、解决方案

1.如果超过了double的范围无法使用double进行保存,只有String才可能准确的保存好这个数据;

2.如果数据是很大的数字要进行数学计算，只能将其变为字符串类型，而后按位取出每一个字符保存的数据，进行手工的计算,相当的麻烦,java考虑到了这种情况，java专门提供了大数字的操作类BigInteger,BigDecimal;

### 三、具体实现

#### 1.BigInteger

java虽然提供了大数字的操作，但是在实际的项目中可能对于数字比较敏感,这时java本身提供的数字类帮助不大;需要使用第三方的进行数学计算的包来实现。如果要使用java提供的数字来一定要考虑计算的结果是否准确;

BigInteger b1=new BigInteger("2355555555333");

BigInteger b2=new BigInteger("1335555555533");

*//加法*

BigInteger add = b1.add(b2);

System.out.println(add);

*//减法*

BigInteger subtract = b1.subtract(b2);

System.out.println(subtract);

*//乘法*

BigInteger multiply = b1.multiply(b2);

System.out.println(multiply);

*//除法 数组里面只有两个元素 第一个元素是商 ，第二个元素是余数*

BigInteger[] result = b1.divideAndRemainder(b2);

System.out.println("商："+result[0]+" 余数"+result[1]);

#### 2.BigDecimal

BigInteger不能保留小数,BigDecimal可以保留小数数据;  主要提供了

public BigDecimal(String val) {}

public BigDecimal(double val) {}*//这个构造可能会产生一些问题（这个**函数返回的结果有些不可预测）*

##### 解决浮点数计算不可预测的问题

如何解决浮点数计算中public BigDecimal(double val){}返回结果不可预测的问题：

1.将结果转换成String类型，再使用BigDecimal（String s）构造方法；

2.使用BigDecimal.valueOf()方法 ；

##### BigDecimal的作用

###### 1.支持基本数学计算并可以准确的实现四舍五入操作；

  a.BigDecimal不仅支持基本的数学计算,且BigDecimal可以用来准确的实现四舍五入；使用Math.round()方法可以实现四舍五入的操作，但是这种操作有一个问题，所有的小数位都四舍五入了(四舍五入取整);比如一家公司的年收入是8.45678亿,按照Math.round()方法相当于只有8亿了;虽然 BigDecimal 没有提供直接的四舍五入的操作支持，但是可以利用除法运算实现;

 public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode) {}

  BigDecimal divisor:除数

  int scale：保留几位小数

  int roundingMode：进位模式;

  public static final int ROUND\_HALF\_UP:四舍五入模式

###### 2.可以解决浮点型数据不能精确计算的问题;

  System.out.println(0.1 +0.2);*//0.30000000000000004*

  System.out.println(0.3 - 0.1);*//0.19999999999999998*

  System.out.println(0.1 \* 0.2);*//0.020000000000000004*

  System.out.println(0.7 / 0.1);*//6.99999999999999*

  原因在于我们的计算机是二进制的。浮点数没有办法使用二进制进行精确表示,可以使用BigDecimal解决

###### 3. 解决BigDecimal(double val)函数返回的结果不可预测问题

问题描述

BigDecimal(double val)函数返回的结果可能不可预测

BigDecimal bd1=new BigDecimal("2.4");*//2.4*

BigDecimal bd2=new BigDecimal(2);*//2*

*//The results of this constructor can be somewhat unpredictable.（这个函数返回的结果有些不可预测）*

BigDecimal bd3=new BigDecimal(2.4);*//2.399999999999999911182158029987476766109466552734375*

System.out.println(bd1+"\t\n"+bd2+"\t\n"+bd3);

解决方案

1.将结果转换成String类型,将结果转换成String类型

BigDecimal bd4=new BigDecimal(Double.toString(2.4));*//2.4*

System.out.println(bd4);

2.将结果转换成String类型

BigDecimal bd5=BigDecimal.*valueOf*(2.4);*//2.4*

System.out.println(bd5);

###### 4.使用BigDecimal实现计算

BigDecimal a=new BigDecimal(Double.toString(0.1));

BigDecimal b=new BigDecimal(Double.toString(0.2));

*//加法*

BigDecimal add = a.add(b);

System.out.println("a+b: "+add);*//0.3*

*//减法*

BigDecimal subtract = b.subtract(a);

System.out.println("b-a: "+subtract);*//0.1*

*//乘法*

BigDecimal multiply = a.multiply(b);

System.out.println("a\*b: "+multiply);  *//0.02*

*//除法(当不能整除的的时候 divide()会抛异常java.lang.ArithmeticException)*

BigDecimal divide = b.divide(a);

System.out.println("a/b: "+divide);*//2*

*//当不能整除的的时候 divide()会抛异常java.lang.ArithmeticException*

BigDecimal e=new BigDecimal(Double.toString(0.5));

BigDecimal f=new BigDecimal(Double.toString(0.21));

System.out.println(e.divide(f));*//java.lang.ArithmeticException:Non-terminating decimal expansion; no exact representable decimal result.*

java提供了对应的构造函数来解决

public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode){}

divisor: 除数 ;

scale:小数点后保留位数;

roundingMode：舍入模式

              ROUND\_HALF\_UP 四舍五入

BigDecimal dev = a.divide(b, 2, BigDecimal.ROUND\_HALF\_UP);

#### 简单封装

##### 1.获取两个浮点数之和

*/\*\**

 \* 获取两个浮点之和

 \* @param a 加数

 \* @param b 另一个加数

 \* @return 和

 \*/

public static double add(Double a,Double b) {

 if (null == a) {

  a = 0d;

 }

 if (null == b) {

  b = 0d;

 }

 BigDecimal a1=new BigDecimal(Double.toString(a));

 BigDecimal b1=new BigDecimal(Double.toString(b));

 return a1.add(b1).doubleValue();

}

##### 2.获取两个浮点数的差

*/\*\**

 \* 获取两个浮点的差

 \* @param a 被减数

 \* @param b 减数

 \* @return 差

 \*/

public static double subtract(Double a,Double b) {

 if (null == a) {

  a = 0d;

 }

 if (null == b) {

  b = 0d;

 }

 BigDecimal a1=new BigDecimal(Double.toString(a));

 BigDecimal b1=new BigDecimal(Double.toString(b));

 return a1.subtract(b1).doubleValue();

}

##### 3.获取两个浮点的积

*/\*\**

 \* 获取两个浮点的积

 \* @param a 因数

 \* @param b 因数

 \* @return 积

 \*/

public static double multiply(Double a,Double b) {

 if (null == a) {

  a = 0d;

 }

 if (null == b) {

  b = 0d;

 }

 BigDecimal a1=new BigDecimal(Double.toString(a));

 BigDecimal b1=new BigDecimal(Double.toString(b));

 return a1.multiply(b1).doubleValue();

}

##### 4.获取两个浮点数的商

*/\*\**

 \* 获取两个浮点数的商

 \* @param a 被除数

 \* @param b 除数

 \* @return 商

 \*/

public static double divide(Double a,Double b,int scale) {

 if (null == a) {

  a = 0d;

 }

 if (null == b) {

  b = 0d;

 }

 BigDecimal a1=new BigDecimal(Double.toString(a));

 BigDecimal b1=new BigDecimal(Double.toString(b));

 return a1.divide(b1, scale, BigDecimal.ROUND\_HALF\_UP).doubleValue();

}

##### 5.利用BigDecimall中的除法运算实现准确的四舍五入;

public static double round(double num,int scale) {

  BigDecimal bd1=new BigDecimal(Double.toString(num));

  BigDecimal bd2=new BigDecimal(1);

  return  bd1.divide(bd2, scale, BigDecimal.ROUND\_HALF\_UP).doubleValue();

}

## Java中lambda表达式

### 语法

(parameters) -> expression

或

(parameters) ->{ statements; }

### 2.优缺点

**优点**

1.简洁;

2.易并行计算。尤其适用于遍历结果，循环计算数值或者赋值的时候非常方便;

**缺点:**

1.若不用并行计算，很多时候计算速度没有比传统的 for 循环快。

2.不容易使用debug模式调试。

3.在 lambda语句中直接强制类型转换不方便。

4.不可以在foreach中修改foreach外面的值。

### 3.使用步骤

1.获取一个数据源（source）；

2.数据转换；

1. 执行操作获取想要的结果（每次转换原有 Stream 对象不改变，返回一个新的 Stream 对象（可以有多次转换），这就允许对其操作可以像链条一样排列，变成一个管道）

### 4.生成 Stream Source 的方式

**1.Collection.stream();**

String [] strArray = new String[] {"a", "b", "c"};

  List<String> list = Arrays.asList(strArray);

  Stream stream = list.stream();

**2.Collection.parallelStream();**

String [] strArray = new String[] {"a", "b", "c"};

List<String> list = Arrays.asList(strArray);

Stream<String> parallelStream = list.parallelStream();

**3.Arrays.stream(array)or Stream.of();**

String [] strArray = new String[] {"a", "b", "c"};

Stream<String> ofStream1 = Arrays.stream(strArray);

Stream<String> ofStream2 = Stream.of("a", "b", "c");

### 5.数据转换；

**1.Stream.map() 对数据元素进行转换操作；**

*//将List<JavaBean>中的Id转为List<String>的集合*

List<String> strList = stuList.stream()

         .map(Student::getId)

         .map(String::valueOf)

         .collect(Collectors.toList());

### 6.转换结果

**1.Stream.collect();**

*//将List<String>集合转换为指定分隔符的字符串*

String str = strList.stream().collect(Collectors.joining(","));

### 7.Stream和集合、String、Stirng[]之间的转换

**1.Stream-->List**

stream.collect(Collectors.toList());

**2.Stream-->ArrayList**

stream.collect(Collectors.toCollection(ArrayList::new));

**3.Stream-->Set**

stream.collect(Collectors.toSet());

**4.Stream-->Stack**

stream.collect(Collectors.toCollection(Stack::new));

**5.Stream-->String**

stream.collect(Collectors.joining()).toString();

**6.Stream-->String []**

//1.构建Array类型的Stream

String [] strArray = new String[] {"a", "b", "c"};

Stream  stream = Arrays.stream(strArray);

//2.将流转为String[]

String[] strArray1 = (String[]) stream.toArray(String[]::new);

## Lambda表达式的应用-集合转换

### 1.List<Pojo>转为List<String|包装类>

#### 1.)对象属性类型和目标类型一致情况

将List<Student>中的Id属性(Long类型)转为对应的类型的集合List<Long>

List<Long> intList = stuList.stream()

    .map(Student::getId)

    .collect(Collectors.toList());

#### 2.)对象属性类型和目标类型不一致情况

将List<Student>中的Id属性(Long类型)转为List<String>

实现方法：使用map()来转换指定类型

**写法1**

List<String> strList = stuList.stream()

        .map(Student::getId)

        .map(String::valueOf)

        .collect(Collectors.toList());

**写法2**

List<String> str1List = stuList.stream()

        .map(t->String.valueOf(t.getId()))

        .collect(Collectors.toList());

### 2.List<Pojo>转为指定分隔符的字符串

#### 1.)Pojo中属性为不是String类型情况

将List<Student>中的ID属性（Long类型）转为指定分隔符的字符串

**1.使用String中的join();**

String str1 = String.join(",",

        stuList.stream()

       .map(Student::getId)

       .map(String::valueOf)

      .collect(Collectors.toList())

  );

**2.使用stream方式**

String str2 = stuList.stream()

        .map(Student::getId)

        .map(String::valueOf)

        .collect(Collectors.joining(","));

#### 2.)Pojo中属性为是String类型情况

将List<Student>中的name属性（String类型）转为指定分隔符的字符串

**1.使用String中的join();**

String str1 = String.join(",",

        stuList.stream()

       .map(Student::getName)

      .collect(Collectors.toList())

**2.使用stream方式**

String str2 = stuList.stream()

        .map(Student::getId)

        .collect(Collectors.joining(","));

### 3.List<Pojo>转为Map

#### 1.)key的属性类型和目标Key类型一致情况

将List<Person>转为Map, key类型为Id属性对应的类型（Long）,value为Person对象;

Map<Long, Person> map = pList.stream().collect(

  Collectors.toMap(

    Person::getId,

    p -> p,

    (k1,k2)->k1

  )

);

#### 2.)key属性的类型和目标属性类型不一致情况

将List<Person>转为Map 目标属性key为String类型Id,value为Person对象;

Map<String, Person> map2 = pList.stream().collect(

 Collectors.toMap(

  person -> String.valueOf(person.getId()),

  p -> p,

  (k1,k2)->k1

 )

);

map2.forEach((k,v)->System.out.println("key: "+k+" value: "+v));

#### 3.)将List<Pojo>转为Map<String,String>

将List<Person>转为Map key为String类型Id,value为Person对象中的name属性;

value不需类型转换

Map<String, String> map3 = pList.stream().collect(

        Collectors.toMap(

         person -> String.valueOf(person.getId()),

         Person::getName

        )

);

map3.forEach((k,v)->System.out.println("key: "+k+" value: "+v));

value需要类型转换

Map<String, String> map3 = pList.stream().collect(

        Collectors.toMap(

         person -> String.valueOf(person.getId()),

         p->String.valueOf(p.getName())

        )

);

map3.forEach((k,v)->System.out.println("key: "+k+" value: "+v));

### 4.List<String>转为字符串

#### 1.)将List<String>转换为指定分隔符的字符串

String str = strList.stream().collect(Collectors.joining(","));

#### 2.)将List<String>转为一个字符

String str = strList.stream().collect(Collectors.joining()).toString();

### 5.String转换为List<String>

将使用分隔符分隔的字符串转换为List<String>

String ids ="1,2,3,5";

 List<String> collect = Arrays.asList(ids.split(",")).stream().map(s -> s.trim()).collect(Collectors.toList());

### 6.String[]转为List<String>

String [] strArray = new String[] {"a", "b", "c"};

List<String> list = Arrays.asList(strArray);

## Java工具类中的命名

### JDK、Spring、Hutoo工具类中的命名

* JDK自带工具类

Arrays.asList();

Objects.equals();

Collections.sort();

* Spring框架工具类

StringUtils.isEmpty();

CollectionUtils.isEmpty()

FileCopyUtils.copy();

* Hutool工具类

StrUtil.isEmpty();

CollectionUtil.isEmpty();

FileUtil.copy();

总结一下分为三种：

1、JDK主要以操作对象的复数形式命名

2、Spring框架的工具类以对象或用途 + Util复数方式命名

3、Hutool框架的工具类则以对象或用途 + Util单数方式命名

实际项目中的工具类命名方式

StringUtil.isEmpty();

StringUtils.isEmpty();

StringKit.isEmpty();

StringHelper.isEmpty();

StringTool.isEmpty();

StringTools.isEmpty();

# 如何编写工具类

1. 如果工具类中的方法全部时静态方法，那么可以将工具类作为一个 abstract 抽象类。
2. 如果工具类中有异常，请抛出，不要自己去 try-catch.
3. 工具类中的方法非 static 的。那你可以将工具类定义为 final class，考虑到工具类应该不能被继承。在私有化它的构造函数，提供一个单例

# Java常用类库

## Guava

Guava 是一套来自 Google 的核心 Java 库，其中包括新的集合类型（如 multimap 和 multiset）、不可变的集合、图库，以及并发、I/O、散列、缓存、基元、字符串等实用工具！

1.Maven依赖

<dependency>

<groupId>com.google.guava</groupId>

<artifactId>guava</artifactId>

<version>31.1-jre</version>

<type>bundle</type>

</dependency>

2.Guava 的好处：

* 标准化 - Guava 库是由谷歌托管。
* 高效 - 可靠，快速和有效的扩展 JAVA 标准库
* 优化 -Guava 库经过高度的优化。
* 函数式编程 - 增加 JAVA 功能和处理能力。
* 实用程序 - 提供了经常需要在应用程序开发的许多实用程序类。
* 验证 - 提供标准的故障安全验证机制。
* 最佳实践 - 强调最佳的做法。

## Commons Lang

Commons Lang提供一些基础的、通用的操作和处理

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

<version>3.12.0</version>

</dependency>

## Hutool

Hutool 是一个小而全的 Java 工具类库，通过静态方法封装，降低相关[API](https://so.csdn.net/so/search?q=API&spm=1001.2101.3001.7020" \t "_blank)的学习成本，提高工作效率，使 Java 拥有函数式语言般的优雅。

<dependency>

<groupId>cn.hutool</groupId>

<artifactId>hutool-all</artifactId>

<version>5.8.8</version>

</dependency>

# Java IO

java.io包定义了两类流

1.字节流(JDK1.0) inputStream outputStream

2.字符流(JDK1.1) reader   writer



## Java IO中常用类

1. **File**
2. **InputStream**
3. **OutputStream**
4. **Reader**
5. **Writer**

## Java 中IO 常用接口

**Serializable**

## 输入、输出操作步骤

如果要进行输入、输出操作一般

1.通过File类定义一个要操作的路径；

2.通过字节流或字符流的子类对象为父类对象实例化

3.进行数据的读（输入）、写（输出） 操作;

4.数据流属于资源操作，资源操作必须关闭;

## OutputStream

### 简介

1.OutputStram 类是一个专门进行字节数据输出的一个类

public abstract class OutputStream extends Object implements Closeable, Flushable

 1.JDK1.5实现了两个接口Closeable,Flushable

 2.JDK1.7 实现了接口 AutoCloseable

1. OutputStream 提供了了三个输出的方法

1. 输出单个字节 ： public abstract void write(int b) throws IOException;

2.输出部分字节数组：public void write(byte[] b,int off,int len)throws IOException

3.输出全部字节数组public void write(byte[] b)throws IOException

3.OutputStream本身是一个抽象类,如果想要对抽象类进行实例化操作,必须借助抽象类的子类来完成; 进行文件的操作可以使用FileOutputStream,提供了相关构造函数；

public class FileOutputStream extends OutputStream

a. 创建或覆盖已有文件public FileOutputStream(File file)throws FileNotFoundException

b.文件内容的追加public FileOutputStream(File file, boolean append)throws FileNotFoundException

### 实例：

//1.定义输出文件的路径

File file = **new** File("D:\\temp\\txt\\a.txt");

//2.如果目录不存在创建目录

**if**(!file.getParentFile().exists()) {//文件不存在

file.getParentFile().mkdirs();//创建目录

}

FileOutputStream fos = **new** FileOutputStream(file);

String str = "hello world,/r/n你好，世界。";

//1.输出单个字节

**byte**[] bytes = str.getBytes();

**for** (**int** i = 0; i < bytes.length; i++) {

fos.write(bytes[i]);

}

//2.输出部分字节数组

fos.write(bytes, 0, bytes.length);

//3.输出全部字节数组

fos.write(bytes);

fos.flush();

}

### 缺点:

使用OutputStream输出数据，需要将数据变为字节数组输出，使用起来不是很方便。

## InputStream

### 简介

1.如果程序需要读取操作,可以是用inputstream实现;

public abstract class InputStream extends Object implements Closeable{}

与OuttputStram一样我们不需要考虑此接口的存在;(Closeable里面的功能inputStream和OUtputStram已经实现)

2.InputStream提供了三个输入的方法

1.读取单个字节public abstract int read() throws IOException

返回值：返回读取的字节内容,如果现在已经没有内容了返回-1;

2.将读取的数据保存在字节数组里：public int read(byte[] b) throws IOException

返回值：返回读取的数据长度;如果读取到了结尾返回-1;

3.将读取的数据保存到部分字节数组里public int read(byte[] b, int off,  int len) throws IOException

返回值：读取的部分数据的长度,如果已经读取到结尾返回-1

3. inputStream本身是一个抽象类,如果想要对抽象类进行实例化操作,必须借助抽象类的子类来完成;

public class FileOutputStream extends OutputStream

public FileInputStream(File file) throws FileNotFoundException

### 实例：

**public** **void** inputFile() **throws** IOException {

//1.定义输出文件的路径

File file = **new** File("D:\\temp\\txt\\a.txt");

**if**(file.exists()) {

FileInputStream fis = **new** FileInputStream(file);

//2.使用字节数组读取

**byte** [] buf=**new** **byte**[1024];

**int** len;

**while**((len=fis.read(buf))!=-1) {

System.***out***.println(**new** String(buf, 0, len));

}

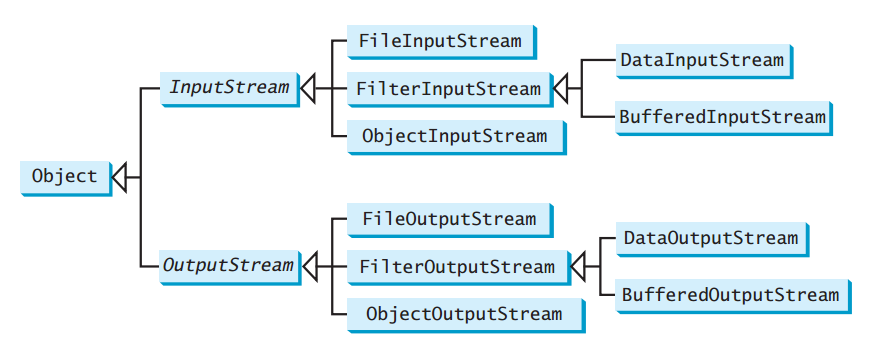
fis.close();

}

}

### 缺点：

**使用OutputStream输出数据，需要将数据（String，boolean ，double,int）变为字节数组然后输出，使用起来不是很方便;**



## Writer

### 简介

1.Writer是JDK1.1之后增加的;除了实现了 Closeable, Flushable接口,还实现了Appendable接口;

public abstract class Writer implements Appendable, Closeable, Flushable{}

public interface Appendable {

    Appendable append(char c) throws IOException;

    Appendable append(CharSequence csq) throws IOException;

    Appendable append(CharSequence csq, int start, int end) throws IOException;

 }

2..在Appendable接口里面定义了追加操作,而且追加的数据都是字符或是字符串; Writer类也提供了以下输入的的方法（部分）

public void write(int c) throws IOException {}*//int——>byte/char ch-->int 所以传int类型比较方便;*

输出全部的的字符数组：public void write(char cbuf[]) throws IOException {}

输出字符串：public void write(String str) throws IOException {}

3.Writer是一个抽象类，如果要实例化操作Writer,需要借助子类对象来实例化(FileWriter);

public FileWriter(String fileName) throws IOException {}

ublic FileWriter(String fileName, boolean append) throws IOException {}

Writer 作为输出流可以进行字符串之间的输出;

### 缺点：

## Reader

### 简介

Reader作为一个进行字符串读取的输入流,本身也是一个抽象类;

public abstract class Reader implements Readable, Closeable {}

public interface Readable {*//jdk1.5添加的;*

    public int read(java.nio.CharBuffer cb) throws IOException;

}

在Reader类总提供了一系列的读取的方法

public int read() throws IOException {}

读取字符串到字符数组:public int read(char cbuf[]) throws IOException {}

返回值：返回数据读取的长度，如果已经读取到结尾了返回-1;

 Reader类实例化可以使用FileReader子类来完成;

## 字符流和字节流的区别

a.字节流直接和终端进行数据的交互,字符流需要将数据经过缓冲区处理后才可以输出;

b.在使用OutputStream输出数据的时候即使没有关闭输出流，内容也可以正常输出;在使用FileWriter输出数据的时候如果没有关闭输出流,那么在缓冲区之中处理的内容不会被强制性的清空,所有就不会输出数据; 如果有特殊情况不能关闭输出流，可以使用flush()强制清空缓冲区内容;

c.在实际开发中使用字节流的情况比较多：图片，音乐，电,对于字符流最大的好处是可以进行中文的有效处理， 如果要考虑中文的时候优先考虑字符流，如果没有中文问题建议使用字节流;

## 转换流

### 简介

java提供了InputStreamReader 和OutputStreamWriter

public class InputStreamReader extends Reader {

  public InputStreamReader(InputStream in) {}

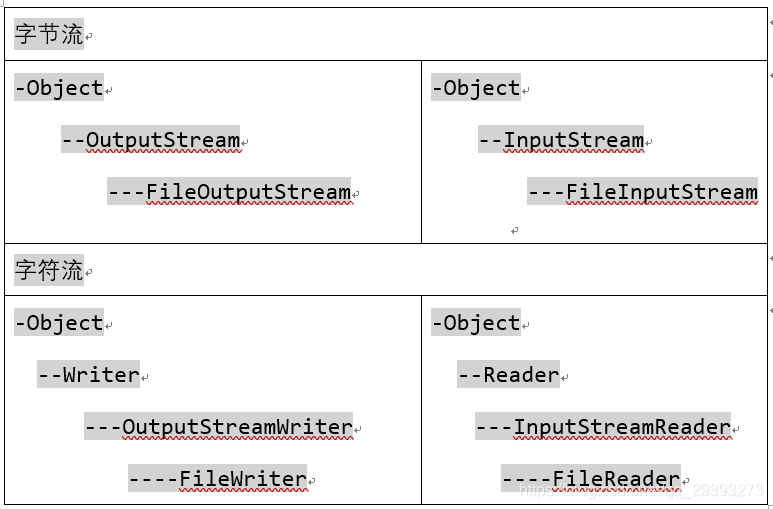
 }

public class OutputStreamWriter extends Writer {

 public OutputStreamWriter(OutputStream out) {}

 }

1. 文件保存在磁盘上，磁盘上能够保存的文件形式都是以字节的方式保存的,在使用字符流读取的时候，实际上也是针对字节数组进行读取，只不过这个转换过程被我们的系统隐藏了,在缓冲区里完成了;
2. 如果要进行转换唯一的可能是处理中文.
3. 两个转换类都是字符流的子类，属于字符流和字节流沟通的桥梁;



## **内存流**

### 简介：

java 提供了两组操作:

字节内存流ByteArrayInputStream和ByteArrayOutputStream

字符内存流CharArrayReader和CharArrayWriter

### 使用场景

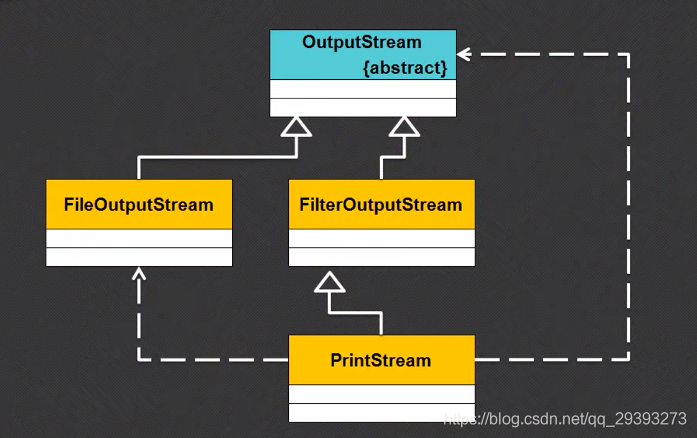
1. **如果某个应用需要进行IO操作,但是不想产生文件的时候，就可以利用内存流实现输入输出的操作;**
2. 利用ByteArrayOutputStream可以实现多个文件同时读取;(实现文件的合并读取)

## 打印流

使用OutputStream输出数据，需要将数据变为字节数组输出，使用起来不是很方便;为了解决使用OutputStream输出数据的不足,java提供了一套专门输出数据的类PrintStream（打印字节流）和PrintWriter（打印字符流）;

public class PrintStream extends FilterOutputStream implements Appendable, Closeable{}

public class PrintWriter extends Writer {}



1.在PrintWriter类中提供了一系列的print()和println();支持各种类型的输出,不再使用write();

2.在整个的操作过程中，虽然操作的形式不同，但是本质上仍然是基于Output类的方法完成的;

3.这种设计模式，java中叫装饰模式;相当于将一个功能不足的操作类,通过某些类的包装，形成更好用的工具类;

4.在实际的开发中，只要是由程序输出内容，都会采用打印流模式完成，但是需要明确的是打印流仍然需要OutputStream的支持;

## 缓冲流

### 简介

Java提供了两组操作：

1. 字符缓冲流 BufferedReader 和BufferedWriter

2.字节缓冲流 BufferedInputStream 和 BufferedOutputStream

在给出的缓冲区数据输入流有两个，其中最重要的是BufferedReader，因为在BufferedReader中提供了一个重要的读取方法：

 public String readLine() throws IOException {}*//读取一行数据一换行为界*

    public class BufferedReader extends Reader {

        public BufferedReader(Reader in){}

    }

如果想使用BufferedReader来处理System.in的操作就比较麻烦了，因为System.in是InputStream类型的;可以利用InputStreamReader来进行转换，且得到的是String类型的数据; 因为String类型的数据可以利用正则进行校验，比较方便；

### 缓冲流作用

1. 解决数据乱码的问题;

2.解决system.in带来的问题;

注意事项：

1.与直接使用[Inputstream](https://so.csdn.net/so/search?q=Inputstream&spm=1001.2101.3001.7020)和Reader类相比使用bufferedReader在进行文件的信息读取的时候会更加的方便;  
2.二进制文件不建议这样读取会出现一些问题;

Scanner

java1.5提供了java.util.Scanner类,来负责解决所有输入流的操作问题; \*

public final class Scanner implements Iterator<String>, Closeable {}

public Scanner(InputStream source) {}

主要有两大类方法 hasNextXxx()和nextXxx()

hasNextXxx():判断是否有指定的数据;

nextXxx():取出数据;

public boolean hasNext(){}

public 数据类型 next() {}

与BufferedReader相比较Scanner更加的直观和简单;

支持正则表达式

String next(Pattern pattern)

boolean hasNext(Pattern pattern)

实现文件的输入

a.在文件读取的时候需要考虑到分隔符的问题;

Scanner useDelimiter(String pattern)

Scanner useDelimiter(Pattern pattern)

## 总结：

1. InputStream 类的功能不足已经被Scanner解决了;
2. InputStream读取数据的时候只能够按照字符串返回.
3. InputStream读取数据的时候所有的分隔符都是固定存在的
4. Reader类的功能不足被BufferedReader解决了

1.解决数据乱码的问题;  
  2.解决system.in带来的问题;

3.OutputStream类的功能不足被PrintStream解决了

使用OutputStream输出数据，需要将数据（String，boolean ，double,int）变为字节数组然后输出，使用起来不是很方便;  
          PrintStream解决了这个问题;(输出时建议使用打印流实现)

4.Writer类的功能不足被PrintWriter解决了；

1.可以方便的写入各种类型的数据;

